新人技術者のための

ロジカル・シンキング入門 第8回

削って，削って…最適化設計（その1）

冴木 元

限られたハードウェア・リソース上で開発する組み込みシステムにおいて，最適化設計を実施して求められる性能を確保することは必要不可欠な作業である．今回は，CPUの演算量を減らすことにより性能を確保する方法について解説する．

（編集部）

Fさんは，今度発売する新しい携帯ゲーム機の開発に携わっています．このゲーム機は「3Dグラフィックスの立体画像がきめ細かくてきれいであること」を売りにしようとしていました．

しかし，開発も終盤にさしかかったところで，思わぬ事態が生じました．画面の動きがトロトロと遅いのです．これでは，迫力ある画像を楽しむどころか，描画を待つ時間のイライラの方が大きく，とても商品として売れそうにありません．

さらに，場合によっては，プレイ中にゲーム自体がハングアップしてしまうこともあると分かりました．どうやら，処理速度がリアルタイムでは間に合わず，システム全体が動かなくなってしまうのが原因のようでした．元凶は3Dグラフィックスなのではないか，とすぐに疑われました．3D画像は，ポリゴンと呼ばれる小さな三角形で物体の表面を覆うようにして表現します．このポリゴンの数を増やせば増やすほど画像はきれいになりますが，計算量が増えるため，その分描画に時間がかかるようになります．

処理量を費やす部分を切り分けていった結果，まず，3D描画を行うライブラリ関数の性能が不十分であり，最適化を行う余地があることが分かってきました．また，3D画像処理を行うビデオ・チップそのものの性能が不十分であることも分かりました．どのビデオ・チップを採用するかに当たっては，当然各社のベンチマークをとって評価したのですが，評価方法に問題があったようです．そのため，評価時の成績と実際にシステムを組み上げたときの処理性能とでは，かなり差があるらしいことが分かりました．

そうはいっても，今からビデオ・チップを取り換えて開発し直すわけにはいきません．結局，3D描画ライブラリの最適化でなんとかすることになってしまいました．

ゲームの作り自体を変更して3Dグラフィックスの描画を簡素化する手はありそうですが，この手段を用いると商品価値に響いてしまいます．また，そもそも各ゲーム・メーカはある程度開発を進めてしまっており，今さら後には引けません．Fさんたちは苦しい立場に立たされてしまいました[注1]．

● 古くて新しい「最適化」

さて，ここに挙げた例は多少誇張されており，実際にここまで切羽詰まってから性能（処理速度とメモリ使用量）の問題がクローズアップされることはまれであると思われます．とはいうものの，組み込みシステムの開発において，性能がしばしば問題となることは事実です．多少まし（？）な開発であっても，開発が終盤にさしかかってから処理量をくっている「犯人探し」が始まるケースなら，枚挙にいと

注1：現在の組み込みシステム開発では，3Dグラフィックスは専用のハードウェア・アクセラレータを用いて，大規模なライブラリ環境を使いこなすのが一般的である．本文中で解説した最適化のテクニックが必ずしもそのまま使えるわけではないので，注意していただきたい．

KeyWord　最適化設計，当たり前品質，アセンブリ言語，LOAD，STORE，レジスタ，ソフトウェア・パイプライン

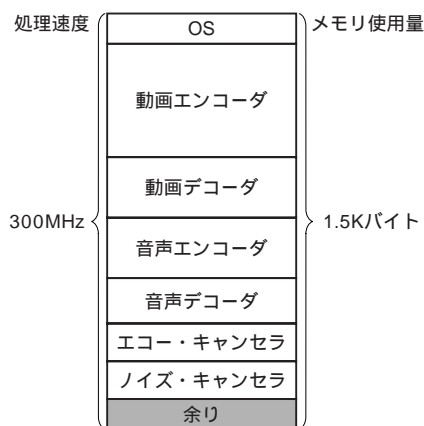



図1　組み込みシステムは性能が万全で当たり前
テレビ電話の組み込みソフトウェアの例．使用するCPUの動作周波数が300MHz，メモリ・サイズが1.5Kバイトだとしたら，ソフトウェアをすべて集めたときにこの範囲に収まるように作らなければならない．これが出来なければ動かないので，商品価値はゼロ．

まがないように思います．

性能問題，裏を返せば「いかにして最適化するか」という，古くて新しい問題が今回のテーマです．

● 性能は「当たり前品質」

最初に注意しなければならないのは，組み込みシステムにおける「性能」とは，多くの場合，備えていて当然（製品が備えるべき最低限の品質である）と認識される性質のものであるということです．例えば，性能を満たしていれば1,000万円で売れるモジュールも，性能が不十分なら500万円にもなりません．それどころか，0円になりかねません（**図1**）．一般的な用語で言い換えると，性能は「魅力的品質」ではなく，「当たり前品質」であるといえます注2．

ソフトウェアは本来，CPUの動作周波数とメモリの大きさの制約を受けるため，処理速度とメモリ使用量にはおのずから限界があります．このこと自体はパソコンで動くアプリケーション・ソフトウェアなどでも変わりはないのですが，パソコンのCPUやメモリは大抵の場合十分にあるので，実際の開発でシビアに性能が問われることはそれほど多くないように思います．

しかし，組み込みソフトウェアの開発では，ハードウェア・リソースがぎりぎりであることがほとんどです．そのため，組み込みソフトウェア・モジュールの開発においては，開発の最初から性能問題を意識する必要があります．最初に全体の工程を考える際に，最適化のフェーズを設けるのはもちろんのこと，後工程での最適化がスムーズに進むように考慮して設計しておく必要があります．

注2：JIS Z9901-1991の定義によると，「当たり前品質」とは，製品が当然備えるべき最低限の品質であり，不十分であるとユーザに不満足感を与える．一方，「魅力的品質」とは，ユーザが製品自体に魅力を感じ満足感を持つ品質を指す．

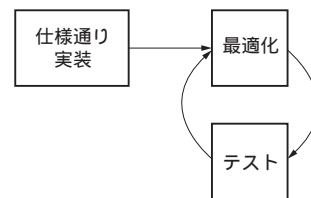



図2　最適化の流れ
最適化するには，その前提として，「正しく動く組み込みモジュール」が存在しなければならない．そして最適化段階のデバッグでは，「正しく動くか」に加えて，「速く小さく作れたか」を考える必要がある．

● 最適化は正しく動くのが大前提

もう一つ注意が必要なのは，最適化というのはあくまでも「仕様通り正しく動くモジュールがある」ことが前提となる，ということです（**図2**）．つまり，最適化をかける以前に，仕様通り動くモジュールが出来上がっている必要があります．そして，最適化した後にも，この品質が損なわれるようなことがあってはなりません．たとえ処理速度が上がっても，その代わりに仕様通り動かなくなってしまったのであれば，最適化とは呼べません（単にバグを埋め込んだことになってしまう）．

従って，最適化のためには，優れた最適化設計の考え方が必要なのはもちろんのこと，品質劣化を防ぐための適切なテスト・ケースが必要となります．

● ボトルネックはCPU？ それともメモリ？

それでは，ここから最適化設計の考え方を解説していきます．「最適化」という言葉は，処理速度の向上とメモリ使用量の削減の両方を意味するものですが，ここでは主として，処理速度の向上に重点を置いて解説します．

最適化設計で重要となるのは，ボトルネックとなるものが何かをあらかじめ見抜いておいて，そのために必要な対策をあらかじめ考えておくことです．ボトルネックとなりうるものは，大きく分けると，次の二つのいずれかです．

1）CPUの演算量

2）メモリのアクセス速度

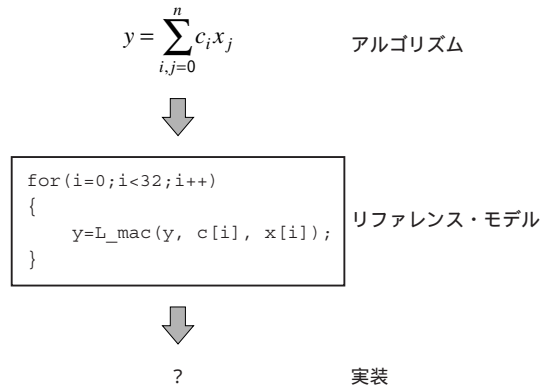



図3　重い演算処理の実装方法がポイント
オーディオ系のコーデックなどは，処理量を費やす演算部分をいかにして最適化するかがポイントとなる．例えば，積和演算がそれに当たるわけだが，開発に用いるCPUのアーキテクチャに照らして，どのように実装すると高速になるかを理解する．

図4　処理速度向上のためのアプローチ
すべての最適化がアセンブリ記述で解決するものではない．メモリの使い方を工夫しなければならないこともある．アセンブリ記述による最適化は，コンパイル結果の流用から始めてハンド・アセンブルに移るとスムーズ．

1) は演算処理自体が多いために，効率の悪い命令で実装していると，処理速度が遅くなってしまうケースです．具体的には，音声系のコーデック処理などで問題となります．これらは，積和演算のような重い処理でループが組まれているため，効率の悪い命令で実装してしまうと，無駄な処理量がループの回数分だけかさんでしまうことになります（**図3**）．

逆に，**2)** のようにメモリのアクセス速度がボトルネックになるのは，演算処理自体の重さよりも，扱うデータ量が大きすぎてメモリ上に配置できない場合です．これは，画像処理などでよくあるケースです．

この連載で何度か述べてきたことですが，CPUコア内部のメモリ（内部メモリ）はアクセス速度が速いのですが，大きさが限られています．そのため，あらゆるデータをそこに配置できるわけではありません．画像処理のように大きなデータを扱う場合は，それらをすべて内部メモリに配置できるわけではないので，外部メモリに配置することになります．しかし，外部メモリは内部メモリに比べて圧倒的にアクセス速度が遅いため，これをいかにして回避するのかが問題となるのです．

今回は，**1)** のケース，すなわちCPUの演算量を削減することによって最適化を図るケースについて，具体的な方針を述べていくことにします．

● 演算速度の向上

CPUの演算量がボトルネックとなるケースの最適化方針を，**図4**に示します．演算処理が重いわけですから，ポイントとなるのは，「仕様を満たす機能の実装をいかにして効率の良い命令に置き換えるのか」です．そのため，最適化の工程においては，C言語のような高級言語で実装されたモジュールをアセンブリ言語などの低級言語に置き換えて実装することが求められるようになります．

組み込みシステムにおけるコンパイラの生成効率は近年飛躍的に向上しています．とはいえ，CPUのアーキテクチャによっては，コンパイラの生成効率に限界があることがあります．そのため，プログラマがコンパイラに代わって最適なアセンブリ・コードを書く必要が生じることがあります．

● まずはコンパイル結果の流用から

さて，「効率的なアセンブリ・コードを書く」といっても，初心者の場合には，アセンブリ言語で一からコーディングするのはなかなか難しいと思います．アセンブリ言語は，C言語のような高級言語とはコーディングのやり方がかなり異なります．ましてや，最近の組み込み開発で用いられるCPUは，一昔前のものと異なり，アーキテクチャが複雑です．従って，命令体系を理解して使いこなすには，なかなか敷居が高い面もあるでしょう．

そこで，初心者が最適化実装をこなすに当たっては，まずはコンパイル結果の改造から着手することをお勧めします．まず，コンパイラにアセンブル結果を生成させて，それを流用することから始めるのです．そして慣れてきたら，今度は一からアセンブリ言語でコーディングすることにします．すなわち，

1）C言語などでコーディング

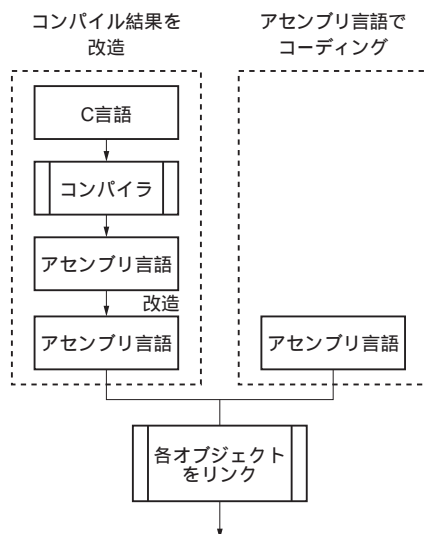



図5　アセンブリ言語によるコーディング
コンパイラが生成するアセンブリ・コードの改造から始める．コンパイル結果の流用では効率の良いコードが書けないモジュールもあるので，一からアセンブリ言語で書く必要も生じる．開発効率や技術の習熟度を考えて工夫するとよい．

図6　アセンブリ言語は演算の前後にLOAD/STOREが必要
アセンブリ言語は高級言語と違って，メモリとレジスタ間でのデータの移動を意識しないとコーディングできない．データはメモリ上に存在するのに対して，演算はレジスタ上で行うからである．

2）コンパイル結果のアセンブリ・コードを改造
3）アセンブリ言語で一からコーディング
という順番で最適化を進めていきます（**図5**）．これは技術を順に習得していく上からも，開発効率の上からも望ましいと思います．もちろん，**1）**の段階で，仕様を満たすように実装する必要があるのは言うまでもありません．

初心者から見ると，CPUの命令解説書の記述から最適なコーディングへとつなげていくのは，なかなか難しいものがあると思われます．そんなときはまず，コンパイル結果のアセンブリ言語を眺めて，その無駄な記述を見破れるようになることで，最適化実装への第1歩とすることができます．

● コンパイル結果の無駄の見抜き方

では次に，コンパイル結果の無駄を見抜くコツについて考えます．

アセンブリ言語による最適化実装というのは，当然のことながら個々のCPUのアーキテクチャに依存したものとなります．そのため，個々の最適化技術は専門技術性が高く，一般論を述べるには難しい面があります．

しかし，「コンパイラがコンパイルした結果を流用する」という前提に立てば，その無駄を省いて最適化を図る際には，個々のCPUのアーキテクチャに依存しない一定のコツが存在します．

コンパイル結果の無駄を見抜く方法というのは，ある個所に注目すれば実は意外とシンプルなものなのです．それは，

1）余計なLOAD/STORE命令を省く
2）ループの外に命令を追い出す
というものです．順に解説していくことにしましょう．

● 余計なLOAD/STORE命令を省く

アセンブリ言語のような低級言語とC言語のような高級言語について，コーディングに対する考え方の違いを挙げればきりがないでしょう．しかし，最適化する際にポイントとなるのは，「アセンブリ言語はメモリからレジスタにデータを移動させないと演算できない」という点です（**図6**）．

データはメモリに格納されているのに対して，アセンブリ言語の演算命令はレジスタ上で行います．メモリからレジスタにデータを移動させることをLOAD，逆にレジスタからメモリにデータを移動させることをSTOREと呼びます．アセンブリ言語のコーディングでは，必ずこのLOADとSTOREを演算の前後で行っています[注3]．従って，コンパイル結果の無駄を見抜く一つ目のポイントは，このLOAD/STOREに無駄がないかどうかに着目することです（**図7**）．

● レジスタの有効活用が最適化のポイント

ここで，「レジスタの数には上限があるから，どんな場

注3：命令体系によってはLOADとSTOREをともにMOVE命令で記述する場合もある．コーディングの際に用いるニーモニック（疑似命令）の呼び方は個々のCPUによって異なるので注意していただきたい．

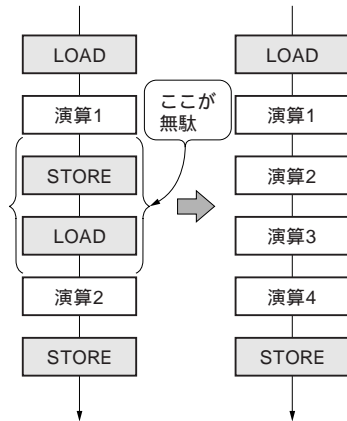



図7　演算の前後のLOAD/STOREに着目
レジスタ上で演算を続けられるのであれば，メモリ-レジスタ間のデータの移動は無駄である．従って，演算の前後にLOAD/STOREが繰り返されていれば，最初と最後に1回ずつLOAD/STOREにまとめればすむことになる．

図8　ループの中は削減効果が高い
ループ中の演算が20サイクルあったとする．ループ回数が200回なら，その部分の処理量は20×200＝4000サイクルとなる．ループ中の処理は削減効果が高いので，最適化する際にはまず，ループの中に注目する．

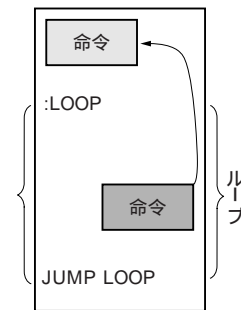



図9　ループの中に注目
コンパイル結果の無駄を見抜くには，ループ中の無駄な命令をループの外にいかにして追い出すかに注目するとよい．ループの外でも実装できる命令は，なるべく外に追い出すこと．具体例を図10，図11に挙げた．

合でもLOAD/STOREが省けるわけではないはずだ」と気付いた方がいるかもしれませんが，全くその通りです．アセンブリ言語レベルでの最適化というのは，「CPUのレジスタをいかに無駄なく活用するか」ということの裏返しでもあるのです．

メモリ-レジスタ間のデータの移動を削減して演算を行えば，演算の途中結果などはすべてレジスタに保持しておかねばならないことになります．CPUが持っているレジスタの数には上限があるため，レジスタの数があふれてしまえば，それ以上はLOAD/STOREが削減できず，メモリ・アクセスが必要ということになります．従って，数に限りのあるレジスタをうまく使い回して余計なLOAD/STOREを演算の間に挟まずにコーディングすることが，アセンブリ言語レベルでの最適化のコツです．

余計なLOAD/STORE命令を削減することは，とりもなおさず，CPUに用意されているレジスタを無駄なく使うことにつながる，ということに注意してください．

● ループの外に命令を追い出す

コンパイル結果の無駄を見抜く方法の二つ目は，「ループの中に注目する」ということです（**図8**）．ループの中の処理は，ループの中の演算量にループ回数を掛け算したものが全体の処理量となります．そのため，ループの中の処理は削減効果が高いので，どのようなアーキテクチャのCPUを用いて実装する場合でも，ループの中の処理をいかにして減らすかというのが，最適化に当たっては常にポイントとなります（**図9**）．

ループの中の処理を工夫によって減らせるものとしては，例えば，テーブルのアドレス計算などがあります．テーブルのようにデータが隣り合わせになっている場合，データをループの中で次々にLOADする場合は，

① 先頭アドレスのLOAD
② オフセットの計算
③ 先頭アドレス＋オフセットの加算
④ ③で求めたアドレスからレジスタへLOAD

という一連の処理を行うことになります．

コンパイル結果を眺めたとき，これらがすべてループの中で行われていたとしたら，最適化の余地ありといえます（**図10**）．これを最適化した具体例を**図11**に示します．LOAD対象のデータが隣り合っている事実に着目することによって，最適化のヒントが得られます．**図10**の③で求

図10
最適化前のアドレス計算
テーブルのように隣り合わせのデータをループの中で次々とLOADする場合，①～④の処理を順にこなすことになる．これらがすべてループの中にあったら非効率的である．いかにして最適化するか？

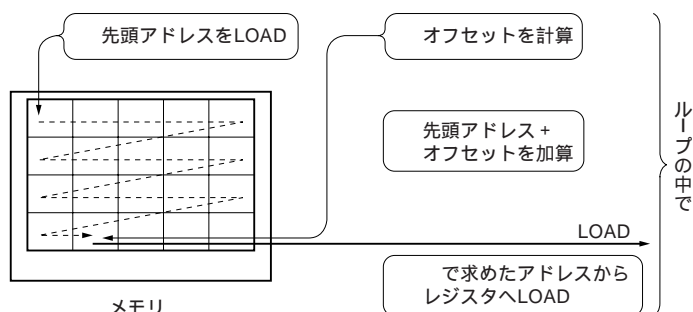



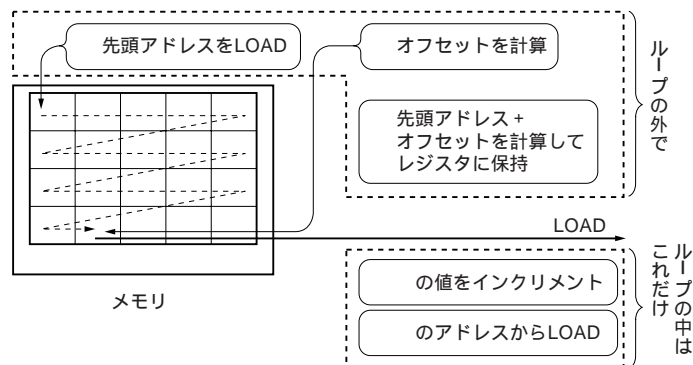
図11
最適化後のアドレス計算
隣り合わせのデータを順にLOADするなら，オフセット値は一つずつインクリメント（増分）すれば求められることになる．ここに注目して，①～⑤のように命令を書き換える．図10と比較すると，ループ中の命令は四つから二つに削減される．

める先頭アドレス＋オフセットという値は，LOADの度に1ずつ増えていくだけです．それならば，このアドレス値をループの外でレジスタに格納しておいて，ループの中でインクリメントしていけばよい，ということになるでしょう．

図10と**図11**を見比べてみてください．この工夫で，ループの中の命令が四つから二つに減りました．

● 最適化にはCPUアーキテクチャの理解が必要

ここまで，コンパイル結果の無駄を見抜いて改造する観点からアセンブリ言語のコーディングを解説してきました．

しかし，「コンパイル結果の流用」というアプローチのみでは限界もあります．アセンブリ言語と高級言語ではデータ構造の持ち方が根本的に異なるため，演算によっては，流用がきかないものもあるからです．例えば，信号処理のフィルタ演算などがそれに当たります．そのような場合には，関数をまるごと一つアセンブリ言語でコーディングする必要が生じます．

最適なアセンブル・コードを書くには，なによりもまず，CPUアーキテクチャの特徴をよく理解する必要があります．そして，自分が実装しようとしている演算を高速化できるような命令がCPUに用意されていないか，CPUのマニュアルをよく読んで工夫する必要があります．

組み込みシステム用に開発されたCPUアーキテクチャは，処理速度を高めるためにいろいろな工夫を施してあります．その特徴を把握することが大切です．

演算効率を高めるためにCPUアーキテクチャが採る方法は，大きく分けて次の二つがあります（実際には，これらの組み合わせでCPUは実装されている）．

1）ソフトウェア・パイプライン

ソフトウェア・パイプラインとは，一連のCPUの命令を並列化して，1サイクルにいくつもの命令を実行できるようにすることです．ロード命令→乗算命令→加算命令→ストア命令という命令を繰り返すのであれば，命令を一つ一つ処理するのではなく，個々の命令を並列化してしまって1サイクルで複数の演算をこなしてしまえば，処理速度は向上することになります（**図12**）．従って，コンパイラがもし命令のソフトウェア・パイプライン化を十分に行っていないのであれば，プログラマがアセンブル・コードを書き換える必要があります．

2）専用命令の実装

専用命令の実装とは，一つの演算器で複数の命令を同時に処理することができるような回路をCPUが内蔵する場合を指します．例えば，積和算を同時に二つ実行できるDual

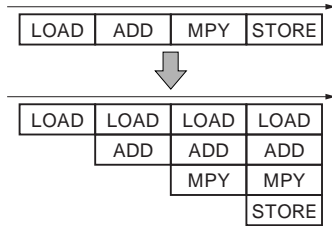



図12　ソフトウェア・パイプラインの例
LOAD，ADD，MPY，STOREと一つ一つ実行するよりも，並列化して次々と実行した方が処理速度は向上する．1サイクルに複数の命令を実行できる機構を大抵のCPUは持っているので，いかにして多くの命令をソフトウェアでパイプライン化できるかが処理速度向上のポイントとなる．

図13　専用命令の例
一つの演算回路が二つの積和算を一度に行えるのがDual Mac命令である．このような専用命令は，使えるレジスタやメモリに制限があることが多いので，制約を満たすようにアセンブリ言語を書くこと．また，そもそもこの命令をコンパイラが生成してくれないこともしばしばある．

Mac命令がCPUに実装されている場合などがそれに当たります(**図13**).

注意しなければいけないのは，このような専用命令がCPUのアーキテクチャには用意されていても，コンパイラはそれを生成してくれないことがむしろ多いということです．その場合は，コンパイラの生成コードとC言語の実装をよく見比べて，効率的な命令をコンパイラが期待通り生成してくれているかをチェックする必要があります．

そして，「自分ならこう書く」というコードをコンパイラが生成してくれていないのであれば，プログラマがアセンブリ言語を自分で書く必要があるということです．

● 最適化ノウハウの汎用性について

CPUの命令を駆使して最適化を行う知識は，専門技術性が高く，個々のCPUに固有のものと従来は考えられてきました．このこと自体は否定の余地がないようにも思えます．しかし，組み込みシステムの開発環境において，技術者たちが培ってきた技量が応用のきかないその場限りのものだ，と考えるのはいかにももったいない気がします．なぜなら，変化の激しい組み込みシステムの開発の現場では，次から次へと新しいアーキテクチャに対応するように迫られることが多いからです．

一つの例として，自動車に組み込まれる車載システムについて考えてみましょう．自動車は，カー・ステレオやカー・ナビゲーションのような運転者の目に見えるものから，エンジン制御のような目に見えないものも含めて，1台の車にさまざまなCPUが積み込まれています．現在では，普通の車でも60個程度，多いものだと100個ものCPUが組み込まれていると言われます．その中で最適化がシビアに問われるものがどの程度あるのかは筆者には分かりませんが，それぞれがエキスパートのみが知る世界で，相互に連絡があり得ないとしたら，寂しい限りのような気がします．

技術者が自分の専門性を深めることは非常に重要です．しかし同時に，自分の技術がどの程度の汎用性を持ちえるのかを考えることも時には必要だと思います．誇り高きベテラン・エンジニアでも，自分が長年親しんできたアーキテクチャが，気が付いたらお払い箱になっていることもあるかもしれません．そうでなくても，やがては管理者として後進の指導に当たらなければならないときに，自分の経験したことだけに閉じた昔話しかできないのだとしたら，開発現場の足をひっぱることになるでしょう．

最適化のように，一見応用のきかない実装依存の知識であっても，汎用化させるとしたらどのようなことが考えられるのかを問うてみるのは，重要であるように思います．

さえき・はじめ

<筆者プロフィール>
冴木 元．システム・エンジニア．「ゴルフのスイングをしながらしゃべるサラリーマン」というのは，ある種典型的な行動パターンと思われるのだが，なぜか筆者の周囲ではお目にかかることが少ない．ゴルフをしないのは単に金がもったいないからか？ 世の中には「ゴルフをするところを見れば人間が分かる」と言う人すらいるらしいのだが，いったい彼らは，何を見てそう言っているのだろう？